

双方向 CTL による Java 最適化器の生成

方 玲[†] 佐々政孝[†]

近年、時相論理によるコンパイラ最適化器生成の研究が行われている。しかし、実際のプログラムを対象として最適化時間や目的コードの実行時間を提示した研究はない。

本研究は従来研究の弱点を克服し、時相論理による記述から Java 言語の効率良い最適化器を生成するシステムを作成した。論理としては、分岐時相論理の 1 つで過去時制と未来時制をとともに扱える双方向 CTL である CTL_{bp} を用いた。

本研究が実装した CTL_{bp} モデル検査器は従来研究と異なり、何の変換も行わずに、過去時制と未来時制を直接扱えるため、効率が大幅に向上した。最適化変換を記す仕様記述も、従来法と異なり、条件式を満たす特定の番号の個々の命令文ではなく、命令文の集合を計算するようにしたので、複雑な最適化が容易に記述できるようになった。さらに、本研究は一時変数を記述できるようにするなど種々の実用化の工夫を行い、Java 言語に対する典型的なコンパイラ最適化器を実現した。

従来、モデル検査器を用いた最適化器は実用的な時間では動作しないとされていたが、実験により、本研究では SPECjvm98 ベンチマークの 7 つすべてについて、15 秒～5 分で最適化が行える事を確認できた。また、生成される最適化器の処理時間を短くするための工夫や、記述のノウハウなど、本手法の改善に向けた種々の考察を加えた。

Generating Java Compiler Optimizers Using Bi-directional CTL

LING FANG[†] and MASATAKA SASSA[†]

There have been several research works that analyze and optimize programs using temporal logic. However, no evaluation of optimization time or execution time of these implementations has been done for any real programming language.

In this paper, we present a system that generates a Java optimizer from specifications in temporal logic. The specification is simpler, and the generated optimizers run more efficiently than previously reported work. We implemented a new model checker for a bidirectional computational tree logic CTL_{bp}, a branching temporal logic. The model checker can check future and past temporal CTL operators symmetrically without any conversion. We also present a new specification language based on the CTL_{bp} that can express typical optimization rules very naturally. By adding rewriting conditions and handling of temporary variables, the system can perform optimization of Java programs. So far, a compiler optimizer using temporal logic was assumed to be impractical, because it consumes too much time. However, with our method, the generated Java compiler optimizer can compile all seven of the SPECjvm98 benchmarks with a compile time from 15 seconds to 5 minutes.

We also gained insights into improving existing techniques for decreasing the compilation time and expertise in specifying compiler optimizations.

1. はじめに

コンパイラ的设计において、コード最適化は重要なパスの 1 つであり、目的コードの時間的・空間的効率を向上させる役割を果たしている¹⁴⁾¹⁰⁾。

最適化器はプログラムを書きかたして作成することがほとんどだが、近年 CTL¹⁹⁾ という論理による最適化の研究も行われている。CTL による最適化は、す

ぐ後で述べる条件付き書換え規則を用いることにより、多くの古典的なコンパイラの最適化を簡潔に表現することができる。

この手法のメリットは、

- 通常最適化器のように数百行ものプログラムを書くことなく、論理式で一行か、多くても十数行で最適化を記述できる。
- 数少ない論理式で書かれるため、解析や証明がしやすい

ということである。

条件付き書換え規則 (conditional rewrite rule) は

[†] 東京工業大学大学院情報理工学研究所

Graduate School of Information Science and Engineering, Tokyo Institute of Technology

「 $I \implies I'$ if ϕ 」の形で記述する．たとえば無用命令除去の例は下記である．

$$\begin{aligned} x := e &\implies skip \\ \text{if } AX((AG \neg use(v)) \vee A \neg use(v) U def(v)) \end{aligned}$$

本研究は分岐時相論理の1つである CTL_{bp} を採用した．本研究での CTL_{bp} は CTL-FV⁵⁾ を基本としており，過去時制が未来時制と対称的に使えるうえ，自由変数が導入されている．

本研究では CTL_{bp} モデル検査器を実装した．従来研究は，NCTL⁸⁾ や μ 計算に変換することで実装しているが，我々の方法では過去時制を未来時制と対称的に検査するため，変換にかかる処理時間を省け，除去によって式が長くなることなく，処理時間が短い．また過去時制に制限が加わることがないので適用範囲も広い．このようにして処理の能力を向上し，従来研究の欠点を克服できた．

また，CTL_{bp} に基づくコンパイラ最適化記述法を提案した．この記述法は実際のプログラム最適化を自然な形で容易に記述できる．

さて，モデル検査を行う前には自由変数を束縛しないとイケない．そのため，自由変数が多い場合は，処理時間が長くなり，現実的ではなくなる．従来研究の記述法は Kripke 構造のノード番号を用いているため自由変数が多く，Kripke 構造のノード番号の束縛に多くの時間がかかる．また，従来方式では多数の命令文が条件式を満たした時，記述が不自然になる．本研究の記述は，従来の方法と異なり Kripke 構造のノード番号を書かなくてよい．モデル検査器は条件式を満たす特定の番号の命令文ではなく，命令文の集合を計算する．そのため，同じ条件式を満たす多数の命令文を書換える処理が容易に記述できるようになった．また，Kripke 構造のノード番号を束縛することを省くことができ，効率を改善した．

最適化記述についてはさらに，書換え条件や一時変数などを扱う処理を加え，これにより典型的な Java 言語コンパイラ最適化器を実現した．

前記のモデル検査器と最適化記述法を用いることにより，本研究は従来研究に比べ，通常の最適化の性能に大幅に近づくことができた．

従来，モデル検査器を用いた最適化器は実用的な時間では動作しないとされていたが，本研究では以上で述べた手法を用いることで，SPECjvm98 ベンチマークの7つすべてについて，15秒から5分程度で動作する最適化器が生成できる事を確認できた．従来研究が処理できない最適化も多く処理できるように

なった．

また，生成される最適化器の処理時間を短くするための工夫や，記述のノウハウなど，本手法の改善に向けた種々の考察を加えた．

筆者の知る限り，通常の最適化器に近づいた性能を持った，時相論理による最適化器の実装は本研究が初めてである．

2. CTL_{bp}

CTL_{bp} は分岐時相論理の1つである．以下，CTL_{bp} とそのプログラム解析と変換での応用について説明する．

2.1 CTL_{bp} とは

CTL_{bp} の提案は文献4)に始まる．過去時制は未来時制とまったく同等，対称的な存在である．時間の構造は各時点が分岐した木構造を対称的に2つ持ち，それぞれ直後と直前の時点を複数個もつものである．CTL_{bp} では CTL の未来時制演算子と対称的な過去時制演算子を持つ．CTL と異なり経路限量子は A, E のほかに $\overline{A}, \overline{E}$ を持ち，それぞれ A, E を逆向きにしたものである．過去時制を制限無しに記述，検証できるため，簡潔性と表現力及び効率が優れている．

本研究での CTL_{bp} は自由変数を導入した CTL-FV⁵⁾ を採用した．

2.2 構文規則

CTL_{bp} の構文規則は以下の通りである．

$$\begin{aligned} CTL_{bp} \ni \phi ::= & \quad | \neg\phi | \phi_1 \wedge \phi_2 \\ & \quad | EX\phi | E\phi_1 U \phi_2 \\ & \quad | \overline{E}X\phi | \overline{E}\phi_1 U \phi_2 \end{aligned}$$

は原子述語である．他の結合子 $EF\phi, EG\phi, AF\phi, \overline{A}X\phi \dots$ は変換によって上記の CTL_{bp} の構文規則の結合子だけを用いた式に変換できる¹⁹⁾．

2.3 意味論

CTL_{bp} の意味論は Kripke 構造によって与えられる．Kripke 構造 K は三つ組 (S, R, L) であり， S は状態の集合， $R \subseteq S \times S$ は遷移関係， $L: S \rightarrow 2^{Pred}$ は各状態にその状態において真となる述語の集合を割り当てる関数である．

K における s_0 からのパスとは， $\forall i \geq 0: (s_i, s_{i+1}) \in R$ となるような状態の (有限もしくは無限の) 列 (s_0, s_1, \dots) である． s_0 からの逆向きのパスとは， $\forall i \geq 0: (s_{i+1}, s_i) \in R$ となるような列である．

論理式 ϕ が Kripke 構造 K の状態 s で真であるという関係を $K, s \models \phi$ で表す．また， K が明らかな場

合には K を省略する．関係 \models は以下のように定義される．

$$\begin{aligned}
s &\models \phi && \text{iff } s \in L(s) \\
s &\models \neg\phi && \text{iff } s \not\models \phi \\
s &\models \phi_1 \wedge \phi_2 && \text{iff } s \models \phi_1 \text{ かつ } s \models \phi_2 \\
s &\models EX\phi && \text{iff } \exists s' ; sRs' \text{ かつ } s' \models \phi \\
s &\models \overline{E}X\phi && \text{iff } \exists s' ; s'Rs \text{ かつ } s' \models \phi \\
s &\models E\phi_1 U \phi_2 && \text{iff ある } s \text{ から始まる経路} \\
&&& s_0 s_1 \dots (s_0 = s) \text{ が存在し, } \exists i \geq 0; s_i \models \phi_2 \\
&&& \text{かつ } 0 \leq \forall j < i; s_j \models \phi_1 \\
s &\models \overline{E}\phi_1 U \phi_2 && \text{iff ある } s \text{ から始まる逆向きの経路} \\
&&& s_0 s_{-1} \dots (s_0 = s) \text{ が存在し, } \exists i \leq 0; s_i \models \phi_2 \\
&&& \text{かつ } i < \forall j \leq 0; s_j \models \phi_1
\end{aligned}$$

CTL_{bp} は開始状態が Kripke 構造の順向き遷移関係から展開した CTL 木と Kripke 構造の逆向き遷移関係から展開した \overline{CTL} 木の二つの木構造を持つ (以後, 明らかな場合は CTL_{bp} をたんに CTL と書くことがある)

図 1 の左は有限 Kripke 構造, 右はその構造を展開した CTL_{bp} 無限木である． s_0 を開始状態として, 実線で書かれた木構造が CTL 木, 点線で書かれた木構造が \overline{CTL} 木である．

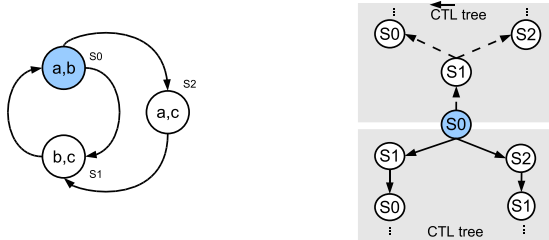


図 1 Kripke 構造とその構造に対応した S_0 からの双方向 CTL 無限木

Fig. 1 Kripke structure (left) and its CTL (infinite) trees for S_0

3. 制御フローモデル

3.1 プログラムの構文

説明のため, ここでは手続きなどがない簡単な命令型言語を考える．

$$= \text{read } X; I_1; I_2; \dots I_{m-1}; \text{write } X$$

ここで, I_1, I_2, \dots, I_{m-1} は文で, 最初の read 文と最後の write 文を含めラベル $n \in \text{Node} = \{0, 1, 2, \dots, m\}$ が付けられている．

命令の BNF を以下に記す．

$$\begin{aligned}
I &::= \text{skip} \mid X := E \mid \text{if } X \text{ goto } n \text{ else } n \\
&\quad \mid \text{goto } n \mid \text{read } X \mid \text{write } X \\
E &::= X \mid E \ O \ E \\
O &::= + \mid - \mid * \mid / \mid \dots \\
X &::= \text{変数名} \\
n &::= 0 \mid 1 \mid 2 \mid \dots \mid m
\end{aligned}$$

本節では, Kripke 構造として制御フローモデルを定義する．

3.2 制御フローモデル

コード に対する制御フローモデルは三つ組 $M = (\text{Node}, \rightarrow, L)$ として定義される．ここで Node は文のラベルの集合であり, 関係 \rightarrow は次のように定義される関係である．以下, 明らかな場合は L を省略する．

$$\begin{aligned}
n_1 &\rightarrow n_2 \text{ iff} \\
& (I_{n_1} \in X := E, \text{skip}, \text{read } X) \wedge n_2 = n_1 + 1 \\
& \vee (I_{n_1} = \text{goto } n \wedge n_2 = n) \\
& \vee (I_{n_1} = \text{if } X \text{ goto } n \text{ else } n' \wedge (n_2 = n \vee n_2 = n')) \\
& \vee (I_{n_1} = \text{write } X \wedge n_2 = n_1)
\end{aligned}$$

この最後の式からわかるように, 最後の文の次のノードは自分自身とする．

$L(n)$ は次のように $n \in \text{Node}$ に対して定義される．

$$\begin{aligned}
L(n) &= \{ \text{stmt}(I_n) \} \\
&\cup \{ \text{def}(x) \mid I_n \text{ は } x := E \text{ あるいは } \text{read } x \text{ の形式である} \} \\
&\cup \{ \text{use}(x) \mid I_n \text{ は } X := E \text{ の形式であり, } x \text{ は } E \text{ で使用される,} \\
&\quad \text{あるいは, } I_n = \text{if } x \text{ goto } n \text{ else } n', \\
&\quad \text{あるいは } I_n = \text{write } x \} \\
&\cup \{ \text{trans}(E) \mid E \text{ は } \text{goto } n \text{ の形式であり, かつ, } E \text{ 中の} \\
&\quad \text{すべての変数 } x \text{ に対して, } I_n \text{ は} \\
&\quad x := E' \text{ あるいは } \text{read } x \text{ の形式} \\
&\quad \text{ではない} \} \\
&\cup \{ \text{entry}(n) \mid n \text{ はプログラムの入口である} \} \\
&\cup \{ \text{exit}(n) \mid n \text{ はプログラムの出口である} \}
\end{aligned}$$

図 2 の左はコードであり, 中はそのコードと対応した制御フローモデル, 右は $L(n)$ のうちの def と use である．

4. CTL_{bp} モデル検査器

この節では, 本研究で実装した双方向モデル検査器について述べる．

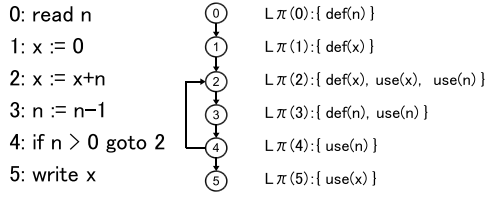


図2 コードと制御フローモデルと原子命題の例

Fig. 2 Example of code, its control flow model and atomic formulas

双方向モデル検査器は従来の研究のモデル検査器の拡張であり、未来時制の検査は既存のアルゴリズムと同じく、CTL木の真理値を求める。過去時制の検査は未来時制を逆向きにし、 \overline{CTL} 木の真理値を求める。

4.1 CTL_{bp} 式の解析

CTL_{bp} 式は木構造で表せる。これを CTL_{bp} 構文木と呼ぶ (CTL_{bp} 木とは異なることに留意)。木の葉は原子述語である。図3(左)の CTL_{bp} 式の構文木を図3(右)に示す。各部分式は表1のようになる。

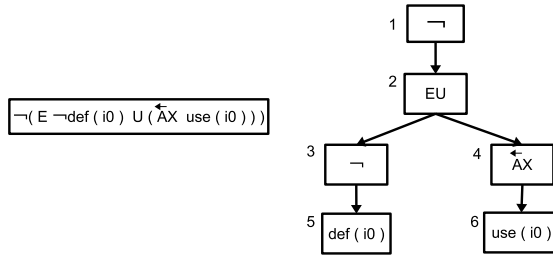


図3 CTL_{bp} 論理式と CTL_{bp} 構文木

Fig. 3 CTL_{bp} formula and CTL_{bp} syntax tree

表1 図3の CTL 構文木の節と部分式

Table 1 CTL_{bp} syntax tree nodes and partial formulas of Figure 3

節	演算子	子	対応した部分式
1	¬	2	$\neg(E \neg \text{def}(i_0) U \overline{AX}(\text{use}(i_0)))$
2	EU	3 4	$E \neg \text{def}(i_0) U \overline{AX}(\text{use}(i_0))$
3	¬	5	$\neg \text{def}(i_0)$
4	\overline{AX}	6	$\overline{AX}(\text{use}(i_0))$
5	ap	def(i ₀)	def(i ₀)
6	ap	use(i ₀)	use(i ₀)

4.2 CTL_{bp} のモデル検査

CTL_{bp} のモデル検査は、CTL 式に対する構文木の葉から根に向かって、対象としている部分式 ϕ_n 毎に各状態 s で満たされるかどうかを計算する。すなわち状態 s_i における部分式 ϕ_n の真理値を $\text{label}(\phi_n, s_i)$ と

するとき、そのラベルの真理値の計算を行う。

$$\text{label}(\phi_n, s_i) = \text{true} \text{ iff } s_i \models \phi_n$$

モデル検査の結果は後の書換え処理に使用されるため、モデル検査をしながら、書換え処理に必要な結果をデータ構造にしまう。このデータ構造は CTL_{bp} 構文木のノードと対応した集合である。

4.3 モデル検査の計算量

プログラムの大きさを命令文の数 n_1 とし、CTL 式の大きさを CTL 構文木のノードの数 n_2 とする。モデル検査は CTL 構文木のすべてのノードですべての状態での真理値を計算するので、

$$\text{モデル検査の計算量} = O(n_1 \times n_2)$$

となる。つまりモデル検査器の計算量はプログラムの大きさと CTL 構文木の大きさに比例する。

5. 最適化の記述

ここでは、本システムでの最適化の記述について説明する。この記述は簡単で自然に通常最適化を記述できる。

5.1 最適化記述の構成

本システムでの最適化の記述は MATCH, CONDITION, PROCESS の3つ部分から成る。MATCH はモデル検査の対象となる命令文の形を示す。CONDITION は命令文が最適化されるべき CTL_{bp} 式を表す。PROCESS は CONDITION の条件式が満たされたとき、どのように変換するかを書く所である。最適化記述は次のような形をしている、くわしくは文献3)を参照。

MATCH

変数 := 式

CONDITION

point_文字列: CTL_{bp} 式

edge_文字列: point_文字列 → point_文字列

PROCESS

point_文字列: コマンド 文

point_文字列: Replace 式 → 式

edge_文字列: EdgeSplit 文

以下は無用命令文除去の最適化記述である：

MATCH

$v := e$

CONDITION

point_delete: $AX((AG \neg \text{use}(v)) \vee A \neg \text{use}(v) U \text{def}(v))$

PROCESS

point_delete: delete $v := e$

MATCH 部は、最適化の対象式の形を決めるとともに、CONDITION 部に現れる自由変数を束縛する。たとえば、

MATCH
 $v := b$

と書くと、右辺が二項式の形の文を対象とする。v は変数、b は二項式である。従って、 $z := x + y$ は対象となるが、 $x := y$ は対象とならない。文 $z := x + y$ が最適化の対象となったとき、 $\{v \mapsto z, b \mapsto x + y\}$ のように書く。集合 $\{v, b\}$ はプログラムの代入文の左辺が変数、右辺が二項式であるときのみ束縛の対象になり、v と b をばらばらに束縛することはない。これにより、例えば $\{v \mapsto x, b \mapsto x + y\}$ 、 $\{v \mapsto y, b \mapsto x + y\}$ のような全組み合わせによる束縛を避けることができる。

CONDITION 部では、条件式とすぐ後で述べる部分式を複数書いたり、それらに式名をつけることができる。

条件式は書換えをするときに成り立つべき条件である。条件式には PROCESS 部での処理と対応させるための式名を付ける。上の例での *point_delete* は式名である。

部分式は長い条件式を書きやすいように、分解して書けるようにするためのものである。条件式に部分式の式名が書かれているときは、その名前が表す論理式に置き換えてモデル検査を行う。この例には現れないが、付録の *point_comp* などは部分式の式名である。

辺についても条件式を書くことができる。辺についての条件式は CTL_{bp} 木構造とは関係がなく、どのような条件式を満たしたノードからどのような条件式を満たしたノードを指しているかを示している。これも CTL_{bp} モデル検査の結果によって計算される。

これらの目的で付けた式名は従来研究でのノード番号と異なり、自由変数ではない。

PROCESS 部には CONDITION 部の条件式を満たした命令文または辺の集合をどのように変換するかを書く。変換を表す処理文に条件式と同じ式名を付けることによって、条件式との対応関係をつける。上の例では、式名 *point_delete* が表す条件式が成り立った命令 “ $v := e$ ” が削除される。必要があればこの部分に一時変数を導入することができる。

一時変数は PROCESS 部の文や式に書くことができるが、CONDITION 部に書くことはできない。

処理文としては文を処理する「コマンド」として *InsertBefore*、*InsertAfter*、*Delete* および *Replace* がある。辺を処理するコマンドとして

EdgeSplit がある。各コマンドの意味は文字通りで、文を前に挿入、後に挿入、削除を行う「*Replace* 式 → 式」は式の一部を書き換える。*EdgeSplit* は辺に文を挿入する。

point ではじまる式名は文を対象とし、*edge* ではじまる式名は辺を対象とする。

さきに、式名は従来研究でのノード番号と異なり、自由変数ではない、と述べた。これについて説明する。

我々のモデル検査器が計算するのは、条件式を満たす特定の番号の文ではなく、条件式を満たす文の集合である。これにより同じ条件式を満たす多数の文を書き換える処理が容易に記述できるようになった。また、後述のように効率を向上させることもできる。この考え方によると、上記の最適化記述には自由変数が $\{v, e\}$ の 2 つある。

一方、従来研究では、無用命令除去を下記のように記述する。

$n : v := e \implies skip$
 $if\ n \models AX((AG \neg use(v)) \vee A \neg use(v) U def(v))$

これは一見して我々の記述と酷似しているが、これには自由変数が $\{v, e, n\}$ の 3 つあり、下記のように、このことは効率に影響する。

従来研究の最適化記述は命令文と対応した Kripke 構造のノード番号を書くことになっているが、このノード番号は自由変数となる。これに対し、本研究の記述は Kripke 構造のノード番号を書かなくてよい。上記の例では、自由変数を 1 つ減らしたことになる。

自由変数の数はシステムの効率に大きく影響する（6 節を参照）ため、自由変数を減らすと、最適化時間が大きく短縮できる。

また、従来研究の最適化記述では Kripke 構造のノード番号を書くことになっているため、いくつかの最適化について、定式化してあるものの、典型的な例で扱えないものがある。

たとえば、共通部分式除去の記述:

$n : (a := e[b]) \implies (a := e[v])$
 $if\ n \models \overline{A}(trans(b) \wedge \neg def(v) U stmt(v := b))$

これは、図 4 の左のケースを扱えるが、右のケースを扱えない。

5.2 最適化の CTL_{bp} による定式化

この節ではコンパイラ最適化の定式化の 2 つの方法について述べる。本研究では、従来定式化を拡張し、データフロー方程式をもとにした定式化も記述できるようになった。

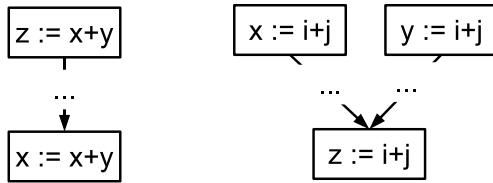


図 4 従来研究が扱える例 (左) と扱えない例 (右)
Fig. 4 Example which can not be processed by previous work

5.2.1 最適化の条件をもとにした方法

これは従来研究の定式化手法と同じである。前述の無用命令除去はこの例である。この方法をもとにして定式化するとき、実際の言語の特徴やモデル検査の効率を考える必要がある。例えば、コピー伝播はコピー元を視点として未来時制を用いて定式化ができるが、コピー先を視点として過去時制を用いて定式化することもできる。このとき、定式化した論理式の自由変数の数や式の長さによって、モデル検査の効率が変わるので、効率の良い定式化方法を選ぶことが望ましい。

5.2.2 データフロー方程式をもとにした方法

時相論理とデータフロー解析との関係は Steffen¹⁷⁾の研究以来指摘されている。我々は実用的な立場からデータフロー方程式を利用した定式化も CTL の枠組で記述できるようにした。

部分冗長性除去は複雑な最適化である。多数の条件式が必要であり、同じ条件式を満たす多数の命令文を書き換える処理をしないと行けない。これを 5.2.1 節のように時相論理の考え方で始めから定式化するのは難しい。

これに対し、データフロー方程式をもとにした定式化は初めから時相論理式を考えるのに比べてはるかに容易である。

本研究ではこの手法を使って、部分冗長性除去を容易に記述できた。部分冗長性除去は長年にわたり研究され、多数のアルゴリズムがあるが、本研究は Paleriらの方法¹³⁾を採用して部分冗長性除去を定式化した。このアルゴリズムは簡明で、変換の結果の計算回数の最適性が証明されている。そのデータフロー方程式に基づいて記述した CTL_{bp} 式を付録に示す。

一般に、データフロー方程式を解くときは、データフロー情報の値が決まる所からはじめて、 $AVIN_0$, $AVOUT_0$, $AVIN_1$, $AVOUT_1$, $AVIN_2$... のように収束するまで繰り返す方法で解いていく事ができるが、CTL 式によるモデル検査では同じように収束するまで繰り返すことができない。そこで、CTL 式に

変換するに当たっては、データフロー方程式のセマンティクスを保ちつつ、繰り返しなしで計算できるような式にするように工夫をし、データフロー方程式と殆ど一対一の形に定式化し、実際の最適化に適用した³⁾。

なお、Lacey の論文⁶⁾の付録にも、同様の考え方が記されているが、マクロを多用し、記述も複雑で、かつ実装されたデータが載っていないため、考え方の提示にとどまっていると思われる。

6. 自由変数

ここでは、自由変数の束縛とその計算量について説明する。自由変数は Lacey⁵⁾の研究で導入された。

自由変数とは、CTL 式の述語に表れて、特定のプログラムの変数とはまだ関係付けられていない変数のことである (自由変数とは論理式で束縛されていない変数、という意味で、プログラムの変数とは異なることに注意) プログラムを扱う際には、自由変数は束縛によって実際のプログラムの変数と関係付ける。たとえば、

MATCH

$v := e$

のような記述では自由変数は $\{v, e\}$ である。仮にプログラムの変数や式が $\{x, y, z, a+b, x+y, -z\}$ であるとすると、自由変数の定義域は

$$\begin{aligned} v &\mapsto \{x, y, z, \dots\} \\ e &\mapsto \{a+b, x+y, -z, \dots\} \end{aligned}$$

である。すなわち、自由変数の定義に忠実に従うと

$$\begin{aligned} \{v ::= x, e ::= a+b\} \\ \{v ::= x, e ::= x+y\} \\ \{v ::= x, e ::= -z\} \\ \{v ::= y, e ::= a+b\} \\ \text{等々} \end{aligned}$$

のように束縛することになる。

自由変数の束縛は全探索になるため、システムの処理時間つまり最適化時間に大きく影響する。

自由変数束縛の計算量

m : CTL 式の自由変数の束縛対象の数 (CTL 式に自由変数 v があった場合、 v と束縛するプログラムの中の変数 x, y, z, \dots の数)

n : CTL 式の自由変数の数

とする、すると

時間計算量: $O(m^n)$

となる。

本研究を含めて、自由変数は多くの従来研究で採用された。これは、自由変数を導入することによって、CTL 式の表現力と便利さが向上するからである。しかし、以上述べたように、自由変数束縛の計算量は大きいので実際の最適化器では自由変数をできるだけ避けるべきである。

本研究では、CONDITION 部に式名を導入することで自由変数を減らすことができた。しかし、MATCH 段階ではすべての自由変数が束縛されてしまう。MATCH 段階での自由変数をどう減らすかは将来の課題である（9.2 節参照）。

7. CTL_{bp} による最適化器

本研究は CTL_{bp} による最適化器を作成した。これは前処理部、モデル検査器と書換え部の 3 部分から構成される。図 5 は本最適化システムの略図である。前処理部は入力プログラムを中間言語の 3 番地コードに替え、自由変数をプログラムの変数や式に束縛する。モデル検査器は前処理部から渡された 3 番地コードと束縛した最適化式を使ってモデル検査を行う。書換え部はモデル検査器で検査した結果に基づき最適化の書換え規則を適用し、コードを出力する。

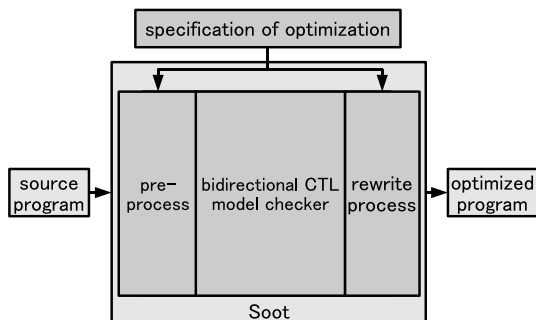


図 5 最適化システム略図

Fig. 5 Outline of the optimization system

システムの実装には Java 最適化フレームワーク Soot¹⁸⁾ を利用した。Soot は新しい最適化処理の開発テスト環境である。追加された新しい最適化処理は、Soot があらかじめ持っている最適化処理（のうちのユーザが指定したもの）に加えて実行される。

図 6 は、左のプログラムに対して、中央の記述から作られた右の CTL 構文木を用いてモデル検査する過程を示したものである。CTL 構文木ではそれぞれの節に式名を満たす集合（対象集合と呼ぶ）の値を付加する。たとえば、右図の構文木の葉から根に向かって、対象としている部分式 $use(i0)$, $def(i0)$, $\overline{AX}(i0)$...

が各状態 $s(0: read\ z0, 1: i0 = 5, 2: i1 = 6 \dots)$ で満たされているかどうかを計算する。書き換えに必要な式名、たとえば $point_a$ の式を満たした状態のノード番号 $\{7, 11, 13\}$ を対応した $point_a$ 集合に入れる。書換え部は $\{7, 11, 13\}$ に対して、式名 $point_a$ で指定された変換を行う。

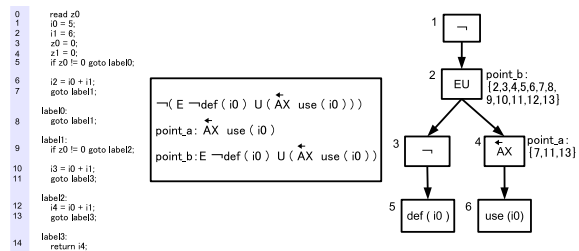


図 6 モデル検査処理

Fig. 6 Model checking

図 7 はモデル検査の結果を使って部分冗長性除去の書換え規則を適用する前と後の例である。

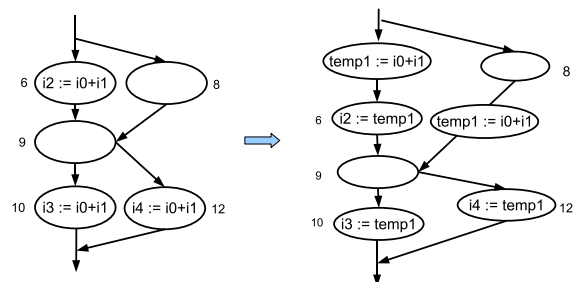


図 7 書換えの例

Fig. 7 Example of rewriting

8. 実験

SPECjvm98¹⁶⁾ 中の 7 つのベンチマークと奥村の Java コード¹²⁾ を使って、実験を行い、データを取得した。

8.1 実験環境

実験環境は下記である。

CPU : Celeron 2GHz

Memory : 512MB

Soot : version 2.2.0

JDK version: 1.5.0_06-b05

計測のオプション : `-Xint -Xms128m -Xmx128m` (JIT とメモリの影響を取り除くため)

適用した最適化 (本研究): 部分冗長性除去 (共通部分式除去とループ不変式も含む)、コピー伝播 (と

定数伝播), 無用命令除去

適用した最適化 (Soot, 比較用): 共通部分式除去, 部分冗長性除去, コピー伝播, 定数伝播と畳み込み, 条件分岐畳み込み, 無用命令除去, 到達不能命令除去, 分岐不能分岐除去, 無用ローカル変数除去

8.2 最適化の処理時間

SPECjvm98 ベンチマークの本研究の手法による最適化時間を表 2 に示す.

表 2 SPECjvm98 の最適化時間 (単位: 秒)

Table 2 The optimization time of the SPECjvm98 benchmark (unit: second)

testcode	compile time
200_check	20
201_compress	29
202_jess	123
209_db	15
213_javac	219
227_mrt	160
228_jack	316

奥村のコードの最適化時間を表 3 に示す.

表 3 奥村のコードの最適化時間 (単位: 秒)

Table 3 The optimization time of Okumura's Java code (unit: second)

testcode	compile time
PiByMachin	0.8
CubeRoot	1.5
Cardano	7.1
CountingSort	2.5
NQueens	3.7
Jacobi	48.6
LogE	20.4
Fibonacci	1.4
Exp	12.1

通常のコンパイラの最適化器ではミリ秒から秒単位で最適化できると考えられるが, 本研究は 15 秒から約 5 分までかかり, 遅い. しかし, 時相論理による最適化は全探索などの処理によって遅くなるのは仕方ないと思われる. しかし後述するように, 本研究は従来研究と比べ, 最適化時間がはるかに短くなった.

8.3 最適化前と最適化後の実行時間の比較

最適化前後の実行時間の比較を図 8, 9 に示す (最適化なしを 1 に正規化した).

本研究は Soot で行う最適化の一部しか適用していないが, それなりの効果を得た. 1 つのベンチマークでは Soot 以上の効果を得た. SPECjvm98 では 202_jess のベンチマークが本手法によって最適化なしに比べて 10% 以上早くなった. 209_db, 213_javac も 2% くら

い効果が出ている. ほかは殆ど変わらない.

この図には, Soot の通常のアプローチにより最適化した結果も載せてあるが, 201_compress, 202_jess, 213_javac 以外は殆ど効果がない.

最適化の效果に影響する原因については 9.1 節で考察する.

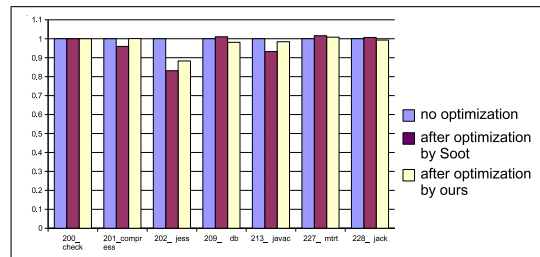


図 8 SPECjvm98 ベンチマーク最適化効果

Fig. 8 Effect of optimization for SPECjvm98 benchmark

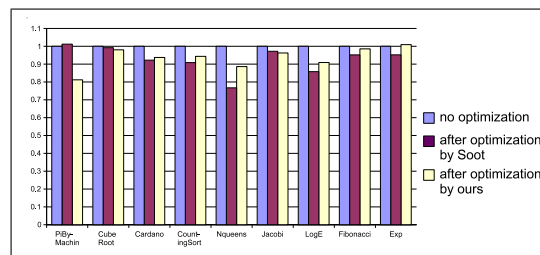


図 9 奥村の Java コードの最適化効果

Fig. 9 Effect of optimization for Okumura's Java code

8.4 Lacey らの手法との比較

Lacey らは最適化時間や実行時間のデータを与えていないので比較できない.

8.5 番らの手法との比較

番らの手法は過去時制を含む NCTL⁸⁾ から過去時制を除去することによってコピー伝播ができるようにしたことが特徴であるため, コピー伝播を適用した最適化時間を比較した. 未来時制のみからなる最適化だと本研究と同じだが, 過去時制を含んだコピー伝播の記述により最適化を行う時間の比較を図 10 に示す (番らの手法による最適化を 1 に正規化した).

この結果, 番らの手法と比べると, 本研究の最適化時間は 15% から 30% と短くなっている.

8.6 同じ最適化を異なる CTL 式で記述したときの最適化時間の比較

図 11 に示したのはコピー伝播を例とし, 異なる CTL 式を我々のシステムに入力し, 自由変数が 2 個 (左) から 4 個 (右) に増えたとき, 計算時間の爆発 (最適

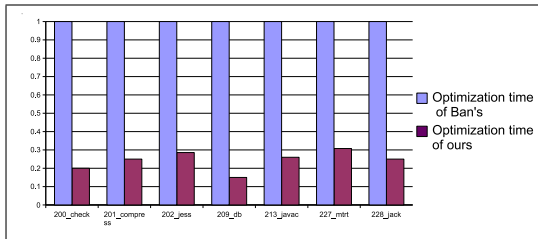


図 10 番らの手法と本研究の最適化時間の比較

Fig.10 Optimization time of our system compared with Ban's work

化時間が 20 秒から 59 分になった) が起こる例である。

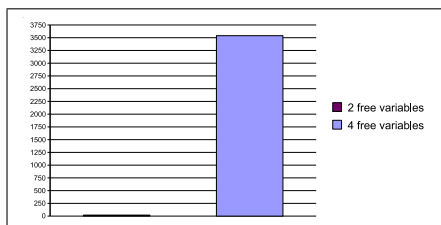


図 11 同じ最適化を異なる CTL 式で記述したときの最適化時間の爆発 (縦軸: 秒)

Fig.11 Example of optimization time explosion (Vertical axis: second)

9. 考察と今後の課題

以下、現在のシステムについて種々考察する。通常の最適化器に近づいた性能を持った、 CTL_{bp} による最適化器の実装は本研究が初めてであるため、その可能性と問題点を明らかにすることが本節の主目的であるので、考察項目が多いことは決して悪いことではないことに留意してほしい。

9.1 考察

9.1.1 CTL の表現力

CTL による最適化は簡単に数行で最適化記述を書け、簡潔であるが、通常最適化アルゴリズムより表現力が劣っている点がある。

細かい処理を CTL 式で書こうとすると、式が長くなるし、式の正しさの証明も難しい。

たとえば、「コピー伝播はもとの命令が無効命令除去により消されるときのみ行う、消されないときは行わない」とか「部分冗長性除去を行うとき、プロファイル情報を用いて計算をよく通るパスからあまり通らないパスに移動したら、正の効果が生じる。これは通常の保守的な最適化ではないが、このような最適化を行いたい」、などを CTL 式で書くのは困難である。

また、複雑なアルゴリズムを用いた最適化を書くことはできない。たとえば、条件付定数伝播²⁰⁾ は、データフロー方程式では定式化できず、表を用いた複雑なアルゴリズムによる最適化である。このような最適化は「モデル検査の結果をすぐ適用しないで途中結果として覚えておく。ある状態に至ったら、適用するか適用しないかを定める」のように書かないといけないが、CTL では記述できない。

また、部分冗長性除去が時間的に最適な結果を得ることを証明するには、CTL 式に基づく証明は難しく、データフロー方程式に基づく証明が現実的である。

9.1.2 最適化器の効率

CTL-FV に基づいた最適化器では自由変数の束縛は CTL の自由変数集合とプログラムの変数集合との全組み合わせになる (6 節参照)。そのうえ、モデル検査は全探索によって行っている。そのため、効率は通常最適化器より遅い。

9.1.3 最適化器の効果

本研究は Java を対象としているため、命令文の移動は例外を超えることはできない。配列、割り算、余算など実行時例外を起こし得るものも全部対象外になるといった制限がある¹¹⁾。

Soot では、BriefGraph というプログラムの制御フローグラフを表すものがある。一方、CompleteGraph というグラフもある。後者は制御フローグラフの上にさらに try 文に囲まれたすべての命令文から catch 文への辺を引いたグラフである、図 12 (左) のプログラムの部分冗長性除去を BriefGraph 上で行った場合の例を図 12 (中) に示す。CompleteGraph 上で行った場合の例を図 12 (右) に示す。図 12 (右) では $x+y$ の移動は太線のパスに抵触するため、保守的な部分冗長性除去のアルゴリズムでは移動を諦めることになる。

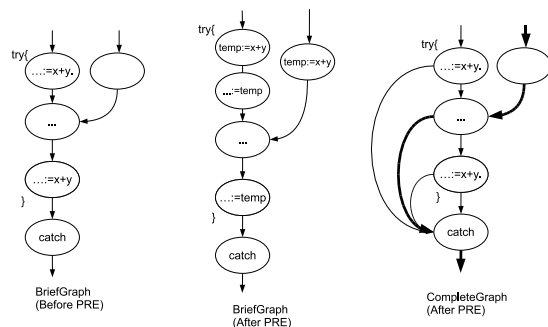


図 12 例外による最適化の阻害

コード移動が例外を超えられない問題は、本研究だけでなく、通常の Java 最適化器にも存在する。

9.2 今後の課題

今後の課題は2つの方向がある。

9.2.1 最適化時間の短縮

モデル検査器を速くするために、BDDを導入したり、部分評価などの手法で、モデル検査器の全探索を避けることが考えられる。

自由変数の束縛はもっとも最適化時間に影響するため、自由変数の数と自由変数の束縛を減らせれば、最適化時間の減少に一番効果的である。

モデル検査の効率と束縛の数を改善できる例は下記のようなものである。次のプログラムを例とする。

```

1:  $x := 100;$ 
2:  $y := 1;$ 
3:  $z := 2;$ 
4:  $w := 3;$ 
5:  $x := z + 1;$ 
6:  $z := x + y;$ 
...
```

- ほかの束縛によって束縛しなくて良い束縛をなくす。たとえば、6: $z := x + y$ の x をコピー伝播の対象とするとき、この x は5: $x := z + 1$ によって代入されているから、さらに前の文1: $x := 100$ と束縛をする必要はない。
- プログラムにない式は束縛しなくてよい。たとえば「 $AX(stmt(v = c))$ 」に対して、素朴には $\{v \mapsto z, c \mapsto 3\}$ も一つの束縛であるが、 $z := 3$ という文はプログラムにないため、このような束縛はしなくてよい。
- 時制経路上にない式と束縛しなくてもよい。過去時制のみの CTL_{bp} 式は過去向きの経路上にある変数だけに束縛し、未来時制のみの CTL_{bp} 式は未来向きの経路上にある変数だけに束縛する。また AX や EX は次の文だけに束縛し、さらに遠い命令文とは束縛しない、などである。

この手法を無用命令除去（順方向のみ）の最適化に適用した実験の結果、最適化の時間はおよそ3分の1になった。今回の研究では時間の関係でこの実装は完成していない。

9.2.2 最適化の効果の向上

最適化の効果を高めるために、例外による最適化の阻害を克服することや、ループ、for文、goto文など細かい解析が必要である。

条件付定数伝播など複雑な最適化をどう書くのかも将来課題である。

双方向 CTL^* という理論体系がある、これを実装

する事によって、もっと強力な表現力をもつ論理式を書ける。こうすると、正確性を保ちやすく、もっと複雑な最適化を書けるようになる。しかし一方モデル検査器の実装は難しくなる可能性がある。これも時間の関係で試していない。

10. 関連研究

従来の研究として、Cousot et al.²⁾ は一番最初に時相論理をプログラム解析に関係づけた。

Laceyらの研究⁵⁾ は、自由変数を導入して拡張した時相論理 $CTL-FV$ を提唱した。伝統的なプログラム最適化の仕様の多くは、 $CTL-FV$ による条件の記述と、その結果を用いた命令文の書換えで記述できることを示した。文献7)、6)の論文はその理論の提案と正当性を述べた。また、いくつかの式について最適化の正しさを証明した。文献6)の論文はこの理論体系を使って最適化する手法の詳細を述べているが、実装については、 μ 計算に変換することによって不動点解を求めるという簡単な説明しかない。最適化時間などのデータもない。また、いくつかの最適化について、定式化してあるが、典型的な例で扱えないものがある。

山岡らの研究²¹⁾ は、既存のモデル検査器 $SMV^{15)}$ を使って実装したが、未来時制しか扱えないうえ、述語は $def()$ と $use()$ しかないため、無用命令除去しか扱えない。

番らの論文¹⁾ は12個の変換式を使って、過去時制を含む $NCTL$ 式を扱えるようにした。過去時制に制限があるうえ、除去処理に時間がかかり、除去によって式が長くなる。その結果、モデル検査の時間がかかり長い。自由変数が多い場合は現実的ではないと予測される。また、最適化仕様にひとつの条件しか書けない、辺を扱えない、などの欠点があるため、無用命令除去とコピー伝播しか行えない。

Lerner et al.⁹⁾ はドメイン指向言語による自動証明付きの最適化器を提唱した。本研究の時相論理による最適化手法とは異なる。

なお、多数の既存研究では定式化にプログラムの文と対応したKripke構造のノード番号を書くことになっている。それらではモデル検査による最適化の効果及び解析の時間について、ベンチマークを使った具体的なデータが提示されていないが、そのような定式化の仕方から束縛の数が爆発すると予測される。

11. まとめ

本研究の主な貢献は次のとおりである。

- CTL_{bp} について、過去時制除去も μ 計算への変

換もせずに直接処理できる効率の良いモデル検査器を実装した。

- 従来研究にあった APPLY_ALL や複雑なマクロなどを使わずに簡潔な形で記述できる最適化記述を提案した。Kripke 構造のノード番号を書かなくてもよくした。モデル検査器は条件式を満たす特定の番号の命令文ではなく、命令文の集合を計算する。そのため、同じ条件式を満たす多数の命令文を書換える処理が容易に記述できるようになった。部分冗長性除去など論理式で書きにくい最適化を定式化できた。
- 実際の最適化処理に欠かせない処理をいくつか加え、実用的な言語である Java 言語を対象として、CTL_{bp} による実用性に近い Java 最適化器を開発した。
- 最適化の時間や最適化の効果について、ベンチマークやテストコードを使ってデータを取得した。その結果、多くの最適化が、実用性に近い時間内で行えることがわかった。
- この経験を通じて、CTL による最適化の問題点をいくつか明らかにすることができた。また各種のデータを提示した。それらの経験とデータはこの分野の研究において重要な参考になると思われる。今後は、問題点を解決し、さらに性能を改善し、CTL_{bp} 式に基づく実用的な Java 最適化器を伝統的な最適化器の一部として組み込めるようにすることを目指したい。

謝 辞

本研究の一部は科学研究費補助金の補助を受けた。

参 考 文 献

- 1) 番 伸宏, 胡 振江, 箕一彦, 武市 正人: Java プログラム最適化の宣言的記述とその効率的な実装, 第 6 回プログラミングおよびプログラミング言語ワークショップ (PPL2004), pp. 65–75, 2004.
- 2) Cousot, P. and Cousot, R.: Automatic synthesis of optimal invariant assertions: mathematical foundations, *SIGPLAN Not.*, Vol. 12, No. 8, pp. 1–12, 1977.
- 3) 方玲: 双方向 CTL による Java 最適化器の生成, 東京工業大学大学院情報理工学研究科数理・計算科学専攻修士論文 2006.
- 4) Kupferman, O. and Pnueli, A.: Once and for all. In *Proceedings of the 10th IEEE Symposium on Logic in Computer Science (LICS 1995)*, pp. 25–35, 1995.
- 5) Lacey, D., Jones, N. D., Van Wyk, E. and Frederiksen, C. C.: Proving correctness of compiler optimizations by temporal logic. In *Proceedings of Symposium on Principles of Programming Languages*, pp. 283–294, 2002.
- 6) Lacey, D.: Program transformation using temporal logic specifications, PhD Thesis, University of Oxford, 2003.
- 7) Lacey, D., Jones, N. D., Van Wyk, E. and Frederiksen, C. C.: Compiler optimization correctness by temporal logic. *Higher-Order and Symbolic Computation*, Vol. 17, No. 3, pp. 173–206, 2004.
- 8) Laroussinie, F. and Schnoebelen, P.: Specification in CTL+Past for verification in CTL. *Information and Computation*, Vol. 156, No. 1/2, pp. 236–263, 2000.
- 9) Lerner, S., Millstein, T., Rice, E., Chambers, C.: Automatically proving the correctness of compiler optimizations, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 364–377, 2005.
- 10) 中田育男. コンパイラの構成と最適化. 朝倉書店, 1999.
- 11) 大平 怜, 平木 敬. 例外依存関係を越える部分冗長性除去. 情報処理学会論文誌:プログラミング, Vol.46, No.SIG 1 (PRO24), pp.134–148, 2005.
- 12) 奥村晴彦:Java によるアルゴリズム事典のソースコード. <http://oku.edu.mie-u.ac.jp/~okumura/Java-algo/>
- 13) Paleri, V. K., Srikant, Y. N. and Shankar, P.: A Simple algorithm for partial redundancy elimination. *ACM SIGPLAN Not.*, Vol. 33, No. 12, pp. 35–43, 1998.
- 14) 佐々 政孝. プログラミング言語処理系. 岩波書店, 1989.
- 15) SMV Model Checker . <http://www.cs.cmu.edu/modelcheck/smv.html>
- 16) SPEC JVM98 Benchmarks . <http://www.spec.org/osg/jvm98>
- 17) Steffen, B.: Generating data flow analysis algorithms from modal specifications, *Science of Computer Programming*, Vol. 21, No. 2, pp. 115–139, 1993.
- 18) Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P. and Sundaresan, V.: Soot — a Java optimization framework, In *Proceedings of CASCON 1999*, pp. 125–135, 1999, <http://www.sable.mcgill.ca/soot/>
- 19) Jan Van Leeuwen (編). 広瀬ほか訳 . コンピュータ基礎論理ハンドブック 形式的モデルと意味論 . 丸善株式会社 .
- 20) Wegman , M.N. , and Zadeck , F.K , Constant

propagation with conditional branches , ACM
Trans. Prog. Lang. Syst. , Vol.13 , No.2 , pp.181–
210 , 1991.

- 21) 山岡裕司, 胡振江, 武市正人, 小川瑞史 . モデル
検査技術を利用したプログラム解析器の生成ツ
ール . 情報処理学会論文誌 , Vol . 44 , No . SIG13
(PRO18) , pp.25-37 , 2003 .

付録 部分冗長性除去の最適化記述

MATCH $v := e$ **CONDITION** $point_comp : use(e) \wedge trans(e)$ $point_avin : \overline{A}X(\overline{A} trans(e) U use(e))$ $point_avout : point_comp \vee (point_avin \wedge trans(e))$ $point_antout : AX(A trans(e) U use(e))$ $point_antin : point_comp \vee (point_antout \wedge trans(e))$ $point_safein : point_avin \vee point_antin$ $point_safeout : point_avout \vee point_antout$ $point_spavin : point_safein \wedge \overline{E}X(\overline{E} (trans(e) \wedge (point_safeout)) U use(e))$ $point_spavout : point_safeout \wedge (point_comp \vee (point_spavin \wedge trans(e)))$ $point_spantout : point_safeout \wedge EX(E (trans(e) \wedge (point_safein)) U use(e))$ $point_spantin : point_safein \wedge (point_comp \vee (point_spantout \wedge trans(e)))$ $point_insert : point_comp \wedge \neg point_spavin \wedge point_spantout$ $point_replace : (point_comp \wedge point_spavin) \vee (point_comp \wedge point_spantout)$ $point_edge1 : point_spavin \wedge point_spantin$ $point_edge2 : \neg point_spavout$ $edge_split : point_edge1 \longrightarrow point_edge2$ **PROCESS** $point_insert : InsertBefore temp := e$ $edge_split : EdgeSplit temp := e$ $point_replace : replace e \rightarrow temp$
